Hardware Support for Accelerating Data Movement in Server Platform

Li Zhao, *Member*, *IEEE*, Laxmi N. Bhuyan, *Fellow*, *IEEE*,
Ravi Iyer, *Member*, *IEEE*, Srihari Makineni, and Donald Newell

Abstract—Data movement (memory copies) is a very common operation during network processing and application execution on servers. The performance of this operation is rather poor on today's microprocessors due to the following aspects: 1) Several long-latency memory accesses are involved because the source and/or the destination are typically in memory, 2) latency hiding techniques, such as out-of-order execution, hardware threading, and prefetching, are not very effective for bulk data movement, and 3) microprocessors move data at register (small) granularity. In this paper, we show this overhead of bulk data movement and propose the use of dedicated copy engines to minimize it. We present a detailed analysis of copy engine architectures along two dimensions: 1) on-die versus off-die and 2) synchronous versus asynchronous. These copy engine architectures are superior to traditional Direct Memory Access (DMA) engines because they are tightly coupled to the core architecture and enable lower overhead communication and signaling. We describe the hardware support required to implement these copy engines and integrate them into server platforms. We perform a detailed case study to evaluate the performance of these copy engines. The evaluation is based on an execution-driven simulator, which was extended with detailed models of copy engines. Our simulation results show that copy engines are effective in reducing the bulk data movement overhead and, hence, hold significant promise for high-performance server platforms.

Index Terms—Copy engine, hardware acceleration, servers, TCP/IP, performance evaluation.

1 Introduction

Data movement (also known as memory copies) is a basic primitive operation that is commonly executed by operating systems, network stacks, Java virtual machines, Web server, database applications, and so forth. The performance of these applications is known to be heavily dependent on the extent to which the memory accesses are overlapped by useful computation. The data movement problem exacerbates the memory latency problem since it requires a train of memory accesses, consumes most of the resources (load/store queues, cache line fill buffers, reorder buffers, and so forth), and stalls the CPU for a long period of time. Several memory latency hiding techniques (out-of-order (OOO) execution [13], multithreading [19], [33], [34], prefetching [5], [7]) that have been investigated apply well to one or a few simultaneous memory accesses but do not address the bulk data movement scenario. In this paper, our aim is to reduce the stall time due to bulk data movement and help improve workloads that are highly dependent on this primitive operation.

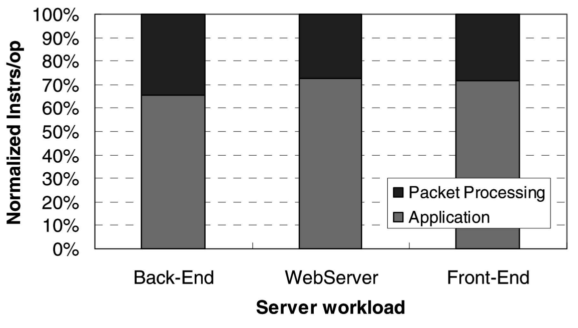
In order to understand the bulk data movement problem, it is important to determine the performance overhead of memory copies in representative workloads. A suitable workload to study memory copy operations is the widely used network protocol stack (Transmission Control Protocol/Internet Protocol (TCP/IP) [23]). A recent study [17] has shown that TCP/IP processing constitutes 30 percent or more of Web server execution time and even database server execution time when using storage over IP [16]. By profiling TCP/IP processing on today's microprocessors, we show that bulk data movement is a significant performance bottleneck. The reasons include the following:

1. Microprocessors move data at register (small) granularity.
2. Several long-latency memory accesses are involved because the source and/or destination are typically in memory (not in cache).
3. The memory accesses clog up all of the memory buffering resources in the CPU.
4. Latency hiding techniques, such as OOO execution, hardware threading, and prefetching, are not very effective.

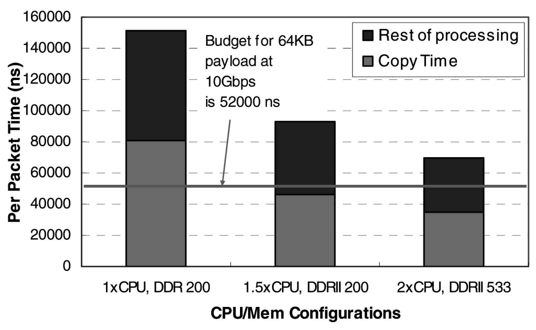
Researchers have attempted to address the bulk data movement solution in the past. From a networking standpoint, the two major solution vectors that have been proposed are copy avoidance [30], [3], [4] and copy acceleration [25], [26]. However, it should be noted that most copy avoidance (zero-copy) techniques have not been adopted widely in commercial operating systems due to their limitations in scope and their specific requirements. For instance, in the case of page remapping [30], when the network packet sizes are smaller than the O/S page sizes, zero-copy is inefficient and requires pages to be pinned down in memory (which, in turn, requires translation look-aside buffers (TLBs) to be flushed). On the other hand, Remote Direct Memory Access (RDMA) [25] achieves zero-copies using TCP Offload Engines

- *L. Zhao, R. Iyer, S. Makineni, and D. Newell are with the Systems Technology Lab, Intel Corporation, MS JF2-58, 2111 NE 25th Ave., Hillsboro, OR 97124.*
  E-mail: {li.zhao, ravishankar.iyer, srihari.makineni, donald.newell}@intel.com.
- *L.N. Bhuyan is with the Computer Science Department, University of California, Riverside, 441, Engineering Building II, 900 University Avenue, Riverside, CA 92521. E-mail: bhuyan@cs.ucr.edu.*

Manuscript received 16 Jan. 2006; revised 19 July 2006; accepted 9 Aug. 2006; published online 26 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0016-0106.
Digital Object Identifier no. 10.1109/TC.2007.1036.



(a)



(b)

Fig. 1. (a) TCP/IP processing overheads. (b) Ten Gbps TCP/IP budget.

(TOEs), which, we believe, are not viable solutions from an economical, as well as a technological, standpoint [21]. Other researchers [3], [4] have proposed new APIs and kernel structures which require modifications to the application and, hence, have not yet been adopted. For copy acceleration, the use of traditional Direct Memory Access (DMA) engines for memory-to-memory copies is quite attractive. However, DMA engines are typically treated as peripheral devices that may impose a significant overhead in communication between the CPU and the DMA engine. Finally, since the DMA engine largely deals with physical addresses (as it has no translation support), user-level applications are not allowed to take advantage of it. Our goal in this paper is to investigate the hardware support needed to enable copy engines (similar to DMA engines) with the requirement that they are tightly coupled into the platform and have low communication overhead.

Our contribution in this paper is the detailed exploration of design and implementation choices for copy engines. These choices are along two vectors: 1) placement of the copy engine with respect to CPU and memory and 2) various modes of operations for the copy engines. Placement of the copy engine is, to a great extent, dictated by the underlying platform architecture. We consider two predominant types of platforms: bus-based centralized-memory (uniform memory access (UMA)) systems and link-based integrated-memory (nonuniform memory access (NUMA)) systems. From a design and performance point of view, we cover trade-offs and point out issues along the following dimensions:

1. proximity to memory,
2. access to cache,
3. interconnect design modifications,
4. coherence protocol changes, and
5. adherence to consistency models.

For the modes of operations, we have considered synchronous and asynchronous execution of copies by the copy engine. We discuss instruction-based triggering of copy execution and evaluate whether the copy execution in the copy engine should be synchronous or asynchronous with respect to the CPU.

Based on our analysis, we propose the most suitable copy engine solutions for the platform architectures considered. We focus on the implementation options and describe the changes required in the platform to integrate the copy engine solution. We model the implementation in an execution-driven simulator and evaluate the performance benefits. Our evaluation is based on a detailed case study of the TCP/IP processing and how using our copy engines helps boost the TCP/IP throughput. Our evaluation shows that the use of a copy engine can speed up TCP/IP processing by 15 percent to 50 percent, depending on the packet sizes processed.

The rest of this paper is organized as follows: In Section 2, we present the bottlenecks in bulk data movement and analyze the limitations that they impose on TCP/IP performance. In Section 3, we present the potential benefits of copy engines if they are introduced in server platforms. In Section 4, we describe the architectural considerations for integrating copy engines in server platforms. Section 5 covers a detailed description of the implementation options for our proposed copy engine solutions. Section 6 presents the evaluation methodology and analyzes the performance benefits of copy engines. Finally, Section 7 summarizes the paper and presents a direction toward future research in this area.

2 The Bulk Data Movement Problem

In this section, we describe why bulk data movement is a problem on server platforms and show its impact on applications by using TCP/IP processing as an example.

2.1 Data Movement Overhead in Network Processing

Although CPU speeds have been increasing at a steady rate, memory speeds and latencies have not kept up and the gap is widening over time. As a result, applications that move lots of data do not scale well with the improving CPU frequency. For instance, copying 64 Kbytes of data from one memory location to another took 80,000 ns on an Intel Pentium M® processor-based server platform with DDR 200 memory with two channels and a 400 million transfers per second front side bus. This forms a significant portion (53 percent) of TCP/IP processing when receiving the same amount of data from the network. Because of this data movement operation, the TCP/IP receive performance scales poorly with CPU speeds. Fig. 1a shows the overhead of TCP/IP processing in data center servers [17] running back-end database workloads (like TPC-C [31]) using storage over IP, Web server workloads (like SPECweb99 [28]), and front end e-commerce servers (like TPC-W [32]). The data shows that TCP/IP processing takes more than

30 percent of the total execution time. Addressing this TCP/IP processing overhead is an active research area [8], [9], [12], [14], [17], [18], especially given the evolution of Ethernet technology from 1 to 10 Gbps and the growing demand for network bandwidth in data centers. The main challenge here is to efficiently scale TCP/IP processing to 10 Gbps speeds so that application can benefit from the increased network bandwidth.

In the network processing context, we show that it is important to solve the data movement problem to scale TCP/IP processing performance. At a 10 Gbps line rate, the TCP/IP stack has a budget of roughly 52,000 ns to receive, process, and deliver 64 Kbytes of data to the application. We have measured and projected TCP/IP receive processing times (including the copy overhead) on current and future server platforms. Fig. 1b illustrates this data by comparing it against the 10 Gbps budget (indicated by the horizontal line in the figure). As processing speed and memory technology improve in the future, we find that the 10 Gbps target budget is not achievable. This is despite the optimistic scaling that we used when projecting the TCP processing time: 1) The measured copy latency was reduced by 42 percent when the memory technology was improved from DDR 200 to DDR-II 400 and beyond and 2) the rest of the TCP/IP processing was linearly scaled with the reductions in the processor cycle time (due to frequency improvements). This scaling is optimistic because TCP/IP processing (even without copies) has several memory accesses that do not scale with the CPU speed. In summary, without significant reduction in the copy overhead, TCP/IP processing at 10 Gbps will be hard to achieve in the near future.

2.2 Related Work

Having identified the need for accelerating data movement, we now cover related work to evaluate whether existing solutions address this problem adequately. Most modern processors implement hardware prefetching [7] and support software prefetching mechanisms [5] to hide memory access latency. However, these mechanisms are not very applicable to bulk data movement. For instance, hardware prefetchers [7] wait for a stride to develop before they begin prefetching: Their impact is limited because the copy is already in progress. Also, they always prefetch more data than is needed as they try to keep ahead of CPU load requests and this results in wasted memory bandwidth. Software prefetching mechanisms [5] enable applications to prefetch a cache line ahead of time. However, most microprocessors do not guarantee prefetch execution, which makes it indeterministic and less appealing for software developers. We have analyzed execution of software prefetch instructions and their impact on data movement. We found that, in most cases, they actually increase the cost of the copy $(\text{prefetch time} + \text{copy time})$. In one case, however, we found a slight (5 percent) reduction in copy time when the copy source is prefetched and no data is evicted from the cache as a result. The underlying reason for this prefetch overhead is that multiple prefetch instructions need to be executed and these take up valuable CPU resources, such as cache line fill buffers and load/store queues.

There are additional techniques that have been proposed to reduce the copy overhead in the context of network processing. These solutions fall into two categories: copy avoidance and copy acceleration. In the copy avoidance category, mechanisms like memory page remapping [30] and RDMA [25] have been proposed. Page remapping [30] has not been implemented in any major commercial operating systems due to associated complexity and limited applicability. On the other hand, RDMA [25] requires TOE hardware support [11], [14] for processing additional protocol layers that are part of RDMA. We do not believe that TOE itself is a viable solution to accelerate TCP/IP processing from an economical, as well as a technological, point of view; our belief is supported by a recent study [21].

In the copy acceleration category, DMA engines [13], [1] have been looked at as a way to perform memory-to-memory copies in addition to the data movement between the conventional Input/Output (I/O) device and memory. However, DMA engines have not entirely succeeded due to the following shortcomings:

1. Descriptor setup entails setting up the [src, dest, length] parameters into shared memory descriptors and adding them to a list that is accessible to the DMA engine. This requires at least one memory access, which costs 300 to 500 clock cycles.
2. Uncacheable triggers that trigger the DMA engine (also referred to as a doorbell) require the use of an uncacheable write to a DMA engine register. Such an uncacheable access typically is a very long latency operation (500 clock cycles).
3. Notification of the copy completion is either through polling or through interrupts; both are expensive, with interrupts being far worse.
4. DMA engines operate in physical address space to prevent the use of the DMA engine by user-level stacks and applications. An alternative is to lock down pages (containing source and destination buffers) in memory, which is prohibitive specifically for use in application space.

Our goal in this study is to find a solution that avoids all of the above overheads and thereby achieve an efficient low-cost asynchronous copy. The copy engine in a server platform is designed to meet the following requirements:

1. low overhead communication between the host processor and the engine,
2. hardware support for allowing the engines to operate asynchronously with respect to processors,
3. hardware support for sharing the virtual address space between the processor and the engine, and
4. low overhead signaling of completion.

Other copy acceleration techniques include the use of larger registers and techniques to improve the memory efficiency by scheduling loads and stores efficiently, as well as bypassing the cache [29]. However, these techniques allow for some speedup of the copy operation. They still stall the CPU for a long time. Our approach, as will be described in Section 3, allows for copy speedup, as well as freeing the CPU to perform overlapping computation. There has also been extensive study to improve the memory performance of applications with irregular access patterns (scatter/gathers, strides, and so forth). One approach is to add new features in the traditional memory controller. Impulse [6] is a smart memory controller that supports application-specified scatter/gather remapping and prefetching. By adding another level of address translation in the memory controller, it allows applications to use the shadow addresses (unused physical addresses) to achieve application-specific operation. It is required to modify the application and the OS, but no modification to the dynamic

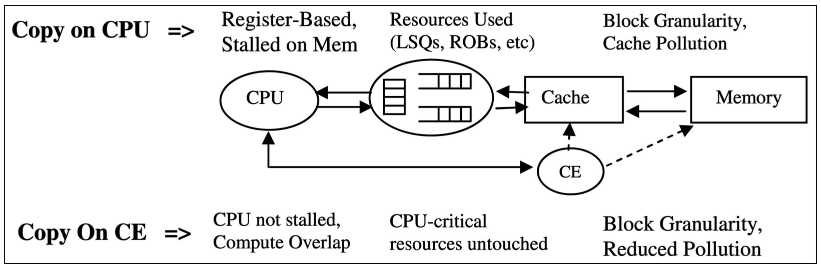

Copy on CPU =>
Register-Based, Stalled on Mem
Resources Used (LSQs, ROBs, etc)
Block Granularity, Cache Pollution
CPU
Cache
Memory
CE
Copy On CE =>
CPU not stalled, Compute Overlap
CPU-critical resources untouched
Block Granularity, Reduced Pollution

Fig. 2. Copy execution characteristics on CPU versus copy engines.

RAM (DRAM) is needed. Another approach is based on Processing-In-Memory (PIM), which integrates processor logic into memory devices [22] so that processors embedded within the memory can take advantage of high bandwidth and low latency to the memory system on the same chip. Our approach here is different because it is much more tightly coupled to the processor execution and requires an instruction set and core enhancements to achieve the benefits.

3 A Case for Copy Engines in Servers

Our analysis in the previous section shows the overhead of executing memory copies in today's platforms and describes why existing latency hiding techniques do not alleviate this overhead. In this section, we introduce the use of dedicated copy engines to accelerate data movement in servers.

Fig. 2 illustrates the basic characteristics of copy execution on CPUs versus copy engines. Performance improvement can be realized by employing copy engines due to the following benefits.

3.1 Potential for Faster Copies and Reducing CPU Resource Occupancy

The memory copy function is usually implemented as a series of load and store operations (memory to register and vice versa). As a result, it ends up occupying several CPU resources and stalls the CPU until the copy completes. Even though the CPU reads data into cache at cache line granularity (64 bytes or higher in most modern processors), it performs copy by reading data into registers that are either 32 or 64 bits long. Copy engines can be used to speed up this copy operation since it can perform copies at higher (cache line) granularity. Another benefit of offloading the copy to a copy engine is that the resources in the CPU are also freed up for other instructions to be executed. For instance, the series of load/store instructions ends up occupying load/store queues, the reorder buffer, and the cache line fill-buffers. As a result, even if the CPU were able to look far ahead in the instruction window and execute other instructions, it would not be able to execute those due to lack of resources.

3.2 Copies Can Be Done in Parallel with CPU Computation

Just like individual memory accesses are overlapped by computation, memory copies can be performed in parallel with CPU computation as well. If asynchronous memory-to-memory copy operations can be enabled (using copy engines), then the CPU is free to perform computation operations. This is similar to a DMA operation, where data is transferred between the memory and the device directly.

3.3 Potential to Avoid Cache Pollution and Reduce Interconnect Traffic

Memory copy is a streaming workload from a caching point of view. Unless the source or the destination is needed by the application after the copy, allocating this data in the cache can result in unnecessary pollution as it may kick out other valuable data from the cache. For many workloads, like the TCP/IP processing, the source of the memory copy is rarely touched by the workload after the copy. However, the destination is touched by the application since it is the recipient of the incoming network data. However, most of the applications (a Web server, for instance) employ multiple processing threads, which may not touch the data immediately or even on the same processor. Thus, allocating the destination may also pollute the cache. The use of a copy engine allows for better control of this pollution, that is, the copy engine can be designed to be configurable so as to allow for various options by the applications running on the server. Similarly, the copy engine can also reduce the interconnect traffic in the platform. For instance, in a system with centralized memory, embedding the copy next to the memory controller can potentially reduce the traffic that is placed on the interconnect (like a shared bus). This has the potential to reduce the queuing delays on the bus and thereby provide additional improvement to the application performance.

Although the copy engine has the potential to provide these benefits, the performance improvement will materialize only if the underlying architecture is carefully considered and the appropriate hardware support is provided in the platform. In Section 4, we will provide a detailed discussion of the aspects that need to be considered before copy engines are implemented.

4 Architectural Considerations for Copy Engines

In this section, we discuss the architectural considerations for integrating copy engines into a server platform. There are two major aspects to consider: 1) the placement of the copy engine and 2) the mode of operation for the copy engine. We start by covering the placement considerations for copy engines.

4.1 Placement Considerations for Copy Engines

The location of the copy engine depends on the underlying platform architecture. We focus our attention on two dominant architectures for server platforms: 1) a bus-based centralized architecture with external memory controllers (EMC [15]) and UMA and 2) a link-based architecture with integrated memory controllers (IMC [2]) and NUMA. Fig. 3

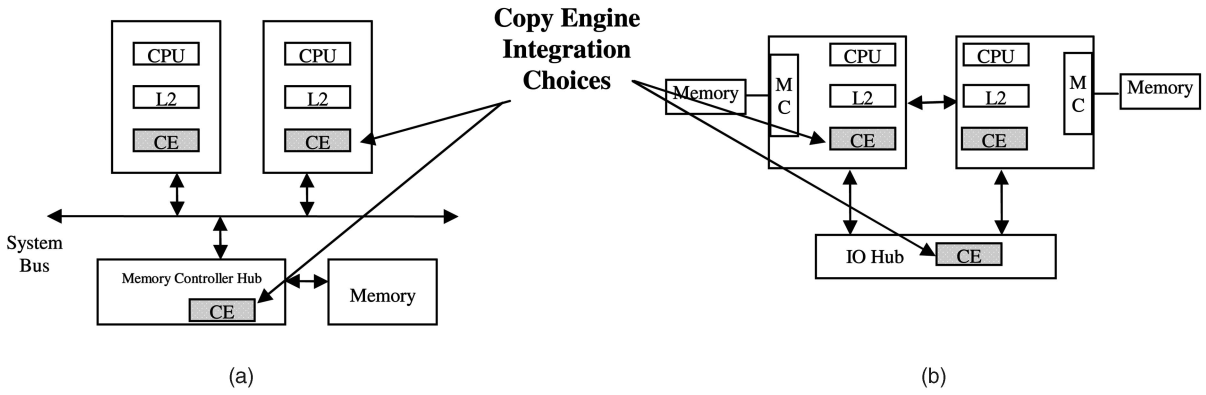


Fig. 3. Copy engine placement in server platforms. (a) UMA/EMC architecture. (b) NUMA/IMC architecture.

illustrates the basic components of these architectures and points out the potential copy engine integration choices.

Since we are discussing the tight coupling of copy engine with the platform (and the CPU implicitly), it should be assumed that the copy engine will be triggered by the execution of a new "*ecpy*" instruction (the design/implementation details will be covered in Section 5). Upon execution of this instruction, the CPU will communicate the parameters of the copy to the copy engine by sending a custom message. This flow is identical for both architectures described in Fig. 3. As shown in the figure, the placement/integration choices for the copy engine can be essentially classified as 1) on-die copy engines and 2) off-die copy engines. The major considerations for integrating on-die versus off-die copy engines in these two architectures are enumerated and discussed as follows:

- **Proximity to memory**. Ideally, a copy engine should be integrated into the memory controller so that it can perform DRAM-aware sequences of reads and writes directly, without occupying any other resources in the platform. In order to be close to the memory controller in the UMA/EMC architecture, the copy engine must be integrated into the memory controller hub (MCH). In the case of NUMA/IMC architectures, memory controllers are integrated into the processors in order to reduce the time to local memory. In this case, the copy engine should be integrated into the processor as well so as to take advantage of the low latency to the local memory subsystem. At the same time, if UMA is desired and one copy engine is to be supported for the entire platform, then the copy engine may be integrated into the I/O hub. However, it should also be noted that, as the number of cores increase on each processor socket (in CMP architectures), it may be also be desirable to provide replicated copy engines on each socket.
- **Proximity to cache**. Another important performance consideration for the copy engine is the access to a cache resource. Since the copy is initiated by the CPU, it is possible that, in some cases, the source of the copy is already in memory. Similarly, it is also desirable in some cases that the destination should be written into the cache by the copy engine so that the application can avoid cache misses when it touches it subsequently. In order to support these caching benefits, the copy engine should have access to the cache. In most architectures (encompassing UMA/EMC and NUMA/IMC), the last level of cache is typically on-die and, as a result, access to cache largely means that the copy engine should be integrated on-die.
- **Interconnect traversal**. Upon execution of the *ecpy* instruction, the CPU communicates the parameters of the copy to the copy engine by sending a request message. As a result, if the copy engine is placed off-die, then additional interconnect support may be required for the message to be appropriately encoded and transmitted to the copy engine. For instance, in the case of an off-die copy engine in the UMA/EMC architecture, the bus protocol will need to support a new transaction that allows an *ecpy* request to be communicated to the MCH and to be passed to the copy engine. In Section 5, we will discuss what this entails in terms of bus protocol design and implementation.

4.2 Operation Modes for Copy Engines

Once the location of the copy engine is decided, the next step is to evaluate the execution modes of the copy. We consider two major execution modes for the copy engine: synchronous and asynchronous.

- **Synchronous copy engine**. The simplest mode for the copy engine is to execute the copy synchronously with respect to the CPU. Here, the copy engine notifies the CPU only after the copy is completed. As a result, the pending instruction that issued the copy to the copy engine will not retire until the copy is completed. Some of the issues to consider here are 1) subsequent instructions that the CPU executes in parallel with the copy engine have no data dependencies (if they have dependencies, then they need to be stalled) and 2) the copy engine is in the cache-coherent domain so that it performs the necessary reads and writes coherently and also listens and responds to other coherent read/write operations that the CPUs perform. One optimization that we design and evaluate for synchronous copy operations is CPU memory request bypassing, explained as follows: Since the copy operation can generate a significant number of reads/writes into the memory queue, the memory requests initiated by the CPUs could be

TABLE 1
Copy Engine Placement Options

	On-die CE	Off-die CE
UMA/EMC	Poor Memory Proximity Good Cache Proximity	Good Memory Proximity Poor Cache Proximity
NUMA/IMC	Good Memory Proximity Good Cache Proximity	Fair Memory Proximity Poor Cache Proximity

stalled for a longer period of time than usual. By allowing CPU requests to bypass the copy engine requests, the CPU can make more forward progress (and overlap computation with the copy).

- **Asynchronous copy engines**. Synchronous copy engines allow the CPU to overlap the execution of as many instructions as can be held in the reorder buffer (which is typically small—around 128 entries). However, the copy operation can take a much longer period of time to complete. Since there is room for additional overlap, we consider asynchronous copy execution by the copy engine. To enable asynchronous copies, the copy engine essentially notifies the CPU of copy completion earlier than the actual completion of the copy operation. To support this, additional hardware support is required to enforce coherence, as well as to serialize subsequent CPU reads/writes based on their dependency on the outstanding copies. For example, once the CPU retires the *ecpy* instruction, the subsequent instructions in the application may attempt to either read/modify the source or destination of the copy. Even worse, the application may free a critical section or lock, which allows another process running on another thread/processor to begin access to the source or destination of the copy. Thus, we require hardware support that allows for dependence checks, thereby ensuring that the load/store requests from the CPU and copy operation by the copy engine are serialized appropriately. Note that the asynchronous copy engine does not require interrupt handling, which is required by DMA engines; thus, it can reduce execution time. In case a context switch occurs, the CPU has to be stalled until the completion signal is received.

5 Design and Implementation of Copy Engines

Having covered the basic considerations and issues for copy engine architectures, we now delve deeper into the design and implementation options for two specific copy engine solutions: 1) off-die copy engines for the UMA/EMC architecture and 2) on-die copy engines for the NUMA/IMC architecture. There are multiple reasons for choosing these two options. As listed in Table 1, we believe these to be the most relevant solutions, given how server platform architectures are evolving. In addition, if we discuss the design and implementation for these, then we can easily apply/extend the solution for the other two possibilities, as shown in the table as well. We cover the design and implementation options for these copy engine solutions (in synchronous and asynchronous modes of operations) by describing the execution flow and pointing out the hardware components affected along the way.

5.1 Triggering Copy Execution on the CPU

In order to trigger copy execution, we now describe the Instruction Set Architecture (ISA) and microarchitectural support required in the CPU. One can notice that the triggering support described is independent of the copy engine type (on-die or off-die) and mode of operation (synchronous and asynchronous). There are three steps involved in triggering the copy engine to start the copy (as illustrated in Fig. 4).

5.1.1 Copy Initiation

A memory copy operation typically requires three operands: the source address, the destination address, and the length of the copy. For a Complex Instruction Set Computer (CISC), one instruction may be enough to specify all three operands and initiate the communication with the copy engine. However, for a Reduced Instruction Set Computer (RISC), more instructions may be required. We assume a RISC machine in order to describe the additional ISA support required. We propose the addition of three new registers (indicated by the "C" prefix to denote Copy). These three copy registers are first initialized with the source (in Cs), the destination (in Cd), and the length (in Cl) by using existing instructions (like *addi*, as shown in Fig. 4). After all of the copy parameters are available, we propose the use of a new instruction called "*ecpy*" to start the process of communicating the copy parameters to the copy engine. At this point, the copy control unit (CCU, shown in Fig. 4) reads the three copy registers and buffers them.

5.1.2 Address Translation

After receiving a copy command, the CCU proceeds to translate the source and destination addresses from a virtual to a physical address space by using the TLB (may require a page walk if a TLB miss is detected). If the memory copy region crosses a page boundary, then this

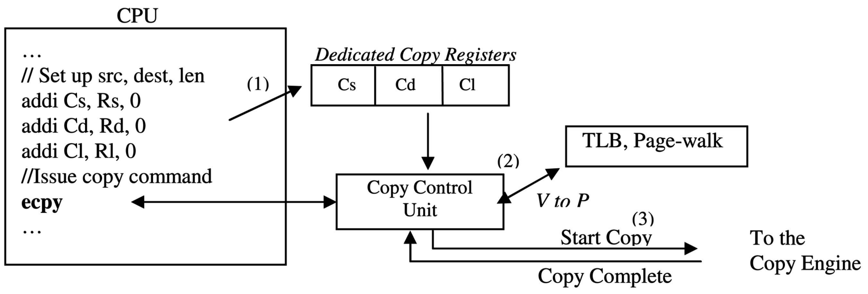


Fig. 4. CPU support for initiating copy operation.

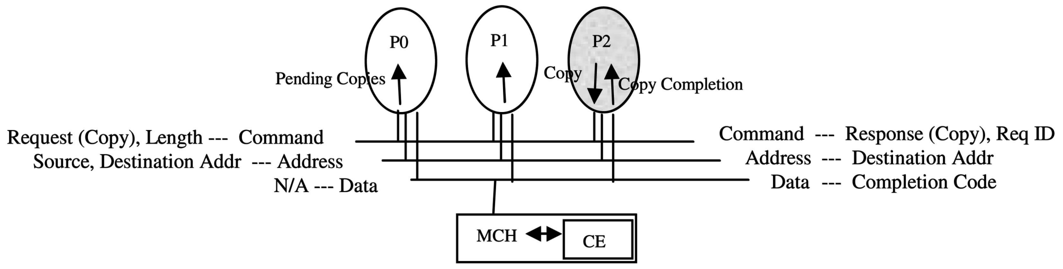

P0
P1
P2
Pending Copies
Copy
Copy Completion
Request (Copy), Length --- Command
Source, Destination Addr --- Address
N/A --- Data
Command --- Response (Copy), Req ID
Address --- Destination Addr
Data --- Completion Code
MCH
CE

Fig. 5. Communication between the CPU and the copy engine.

copy must be split up into several operations, each of which has three operands (source, destination, and length) with contiguous physical memory regions.

5.1.3 Copy Communication

Once the translation(s) is complete, each resulting copy (addresses and length) is individually communicated to the copy engine. It should be noted that the communication of the three parameters to the copy engine should be "atomic" and "ordered" in order to avoid any interleaving of parameters between simultaneous memory copies issued by different processors in the platform. We discuss the interconnect support needed for this in Section 5.2.

5.2 Communication between the CPU and the Copy Engine

The communication between the CPU and the copy engine and back depends on the placement of the copy engine and the architecture under consideration. Here, we describe the communication for the two architectures considered.

5.2.1 Off-Die Copy Engine

In the UMA/EMC architecture, communication between the copy engine and the CPU needs to traverse the global interconnect (a shared bus). A typical pipelined bus [24] consists of address lines, command lines, and data lines. A bus transaction goes through several phases: the phases of interest here include arbitration, request, snoop, response, and data. After the arbitration phase, the CPU is allowed to place a transaction on the bus. The transaction typically consists of the request type (such as read, write, and invalidate), the length of the transaction, and the physical memory address.

For sending a copy transaction on the bus (shown in Fig. 5), we require two addresses (source and destination) and the length of the copy to be placed on the address/command bus. In order to do so, we encode a new request type called *copy*. By placing the copy request type on the bus during the request phase, we indicate that two different addresses will be transmitted to the agents (like CPU and MCH) on the address lines in the subsequent clock cycles. During these clocks, the length of the request is asserted in the command lines. As a result, all nodes in the system are able to latch the request on the bus. The latched copy addresses can be used for two purposes: 1) to perform snoops and 2) to detect dependencies for subsequent reads/writes. The snoop phase may or may not be used, depending on the copy mode of operation. We will discuss this in more detail in Section 5.3.2. The response and data phases are largely ignored for this transaction since it is much like a memory write transaction, where the CPU does not expect any response or data. Once the MCH collects the copy addresses and the length, it communicates the copy command to the copy engine. After some period of time (depending on the mode of operation), the copy engine will place the completion status on the bus so that the completion of the copy is visible to all the nodes in the system. The completion is indicated by placing the address of the copy destination on the bus, the status of the copy on the data bus, and the copy request, as well as the ID of the requesting agent, on the command lines.

5.2.2 On-Die Copy Engine

In the NUMA/IMC architecture, the copy engine is much simpler to implement since the copy engine is on-die and not connected to an external interconnect (like a bus in the previous case). The request can be communicated between the CCU and the copy engine either through point-to-point connections between these units or by enabling a new custom message that is routed by an internal switch. The details of this communication are beyond the scope of this paper.

5.3 Copy Engine Execution Flow

Most of the implementation choices discussed above were independent of the mode of execution chosen for completing the copy transaction itself. In Section 4.2, we introduced the synchronous and asynchronous modes of execution. Here, we describe the implementation of these execution modes. Fig. 6 illustrates the flow of execution starting after the copy command is communicated to the copy engine and ending with the copy engine notifying the CPU of copy completion.

5.3.1 Synchronous Copy Engine

For the synchronous copy execution, the copy engine performs coherent reads and writes by 1) sending out the necessary snoops to all the processors in the platform and 2) performing speculative memory reads/writes. By doing these in parallel, the latencies are overlapped and the overall copy time can be significantly reduced. Once the copy is completed, the copy engine sends a notification to the requesting processor, which then retires the "*ecpy*" instruction.

5.3.2 Asynchronous Copy Engine

As described earlier, the asynchronous copy operation attempts to provide more computation overlap by allowing the *ecpy* instruction to retire even before the copy transaction is completed. To accomplish this, the copy engine essentially needs to make the outstanding copy globally observable by informing all processors that it is using the memory regions (source and destination). Typically, this is

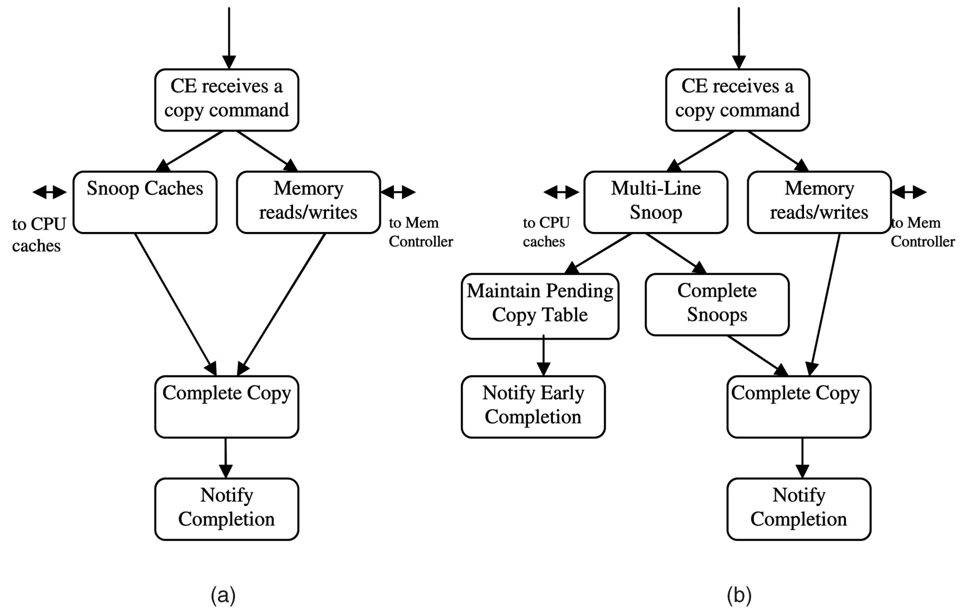


Fig. 6. Copy execution flow: synchronous versus asynchronous. (a) Synchronous copy engines. (b) Asynchronous copy engines.

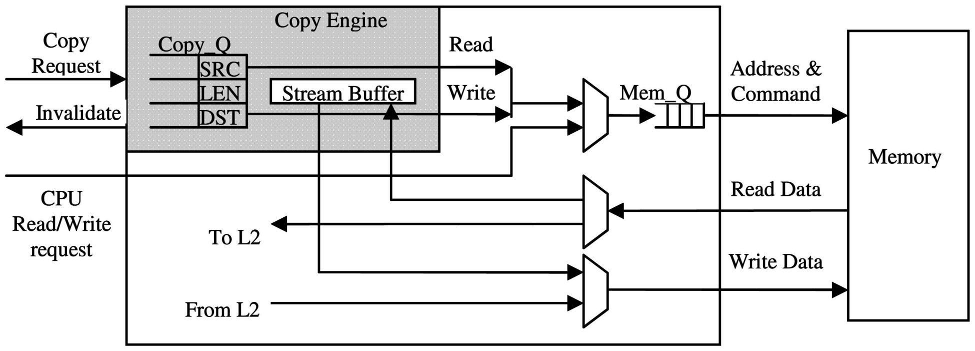


Fig. 7. Block diagram of the memory copy engine.

made possible by broadcasting individual snoop operations. However, we propose the use of a *multiline snoop* operation to reduce the overhead of broadcasting one snoop per cache line in the buffer and receiving individual responses for each. A multiline snoop operation sends the base address and the length of the buffer to the processor cache. Upon receiving the multiline snoop for the source and the destination of the copy, the CCU enters this information in a pending copy table (to track the pending copies) and sends back an acknowledgment to the copy engine. As shown in Fig. 6b, the copy engine then sends an early completion notification to the requesting processor so that it can retire the *ecpy* instruction. All subsequent loads and stores from any processor are locally compared against the entries in the pending copy table and are stalled and retried until the copy is actually completed. Once the copy engine completes the copy, it broadcasts the completion notification to the processors in the platform so that the local CCU can delete the copy entry from the pending copy table. This releases any loads and stores that are pending in the processor.

5.4 Scheduling Memory Transactions

In this subsection, we discuss the design of the copy engine for initiating the reads/writes into the memory subsystem. As shown in Fig. 7, the copy engine consists of a queue of control registers that store the source, the destination addresses, and the length of outstanding copy commands. It also contains a stream buffer to store the copied data. Since the data transfer size typically requested from the memory is one cache line (for example, 64 bytes), more data transfers are required if the copy length is larger than 64 bytes. Assuming that there is one memory queue Mem_Q in the memory controller, the copy engine injects a sequence of read requests to Mem_Q. The copy engine stores data read from the memory stream buffer. Then, the copy engine injects a sequence of write requests to Mem_Q to store the read data at destination address. By issuing a stream of reads and writes, we can exploit the row locality in the memory. It is known that modern DRAM chips latch an entire row (row buffer) on the first access to that row. Subsequent accesses to the same row have much lower latency as a result. This is because subsequent accesses will not incur the precharge and row access times.

The scheduling of CPU and copy engine requests differ, depending on the mode of operation (synchronous versus asynchronous). In the synchronous mode of operation, memory requests from the CPU are not given higher priority over the copy engine-generated ones. When the copy engine is active, it issues several read/write requests into the Mem_Q. As a result of this, the CPU requests may suffer long delays when accessing memory. This limits the

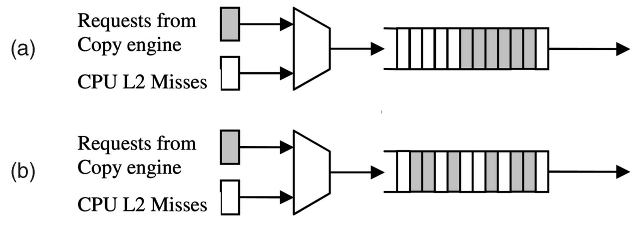


Fig. 8. Memory scheduling approaches.

extent to which the CPU can overlap other computation with the copy operation. To let the CPU make maximum forward progress while the copy engine is busy, we have implemented a CPU request bypass mechanism, where the CPU memory requests are placed in front of copy engine-generated requests in the Mem_Q. Fig. 8 illustrates these two scenarios. To avoid copy engine starvation, we have implemented a fairness algorithm, which guarantees a predetermined amount of memory bandwidth for the copy engine. In the rest of the paper, we use SYNC_WLB to indicate CPU request bypass during the synchronous mode and SYNC_NLB to indicate no bypass during the synchronous mode. ASYNC indicates the asynchronous copy mode, which assumes the CPU request bypass.

5.5 Taking Advantage of Local Cache

For an on-die copy engine, access to local cache can be a significant asset. For instance, the copy engine may find the source of the copy already in the cache of the requesting CPU. As a result, the speed of the copy can be greatly improved by looking up the cache before performing a global coherent operation. Another potential benefit is the invalidation of the source after completion of the copy. Also, instead of writing the destination of the copy into memory, the copy engine can potentially write the data directly into the cache. The trade-offs here are related to 1) the distance between the copy completion and the subsequent access of the destination data, 2) whether this processor touches the data or another, and 3) the overhead of potential cache pollution. In our evaluation, we largely discuss writing of destination data to the memory but compare it to the speedup that can be obtained if the destination is indeed written to cache.

5.6 Copy Retirement and Dependency Checks

These are the last two steps that are required in the CPU to ensure that copies are completed appropriately, and all dependency checks are maintained appropriately.

5.6.1 Copy Completion

As mentioned earlier, the CCU receives copy notification from the copy engine(s) in the platform. It has to deal with three types of copy notification: 1) an early copy completion notification for a copy that it generated, 2) a final copy completion notification for a copy that was initiated either by the final notification itself or by another unit, and 3) a multiline snoop notification for a copy that was initiated elsewhere. When receiving an early notification or a final notification, the CCU ensures that the *ecpy* instruction is retired by the CPU if it initiated the copy. Upon receiving a final notification, it also deletes the copy entry from the pending copy table. Upon receiving a multiline snoop transaction, the CCU basically enters the addresses in the pending copy table so that it can enable dependency checks for asynchronous copy execution (as illustrated in Fig. 9).

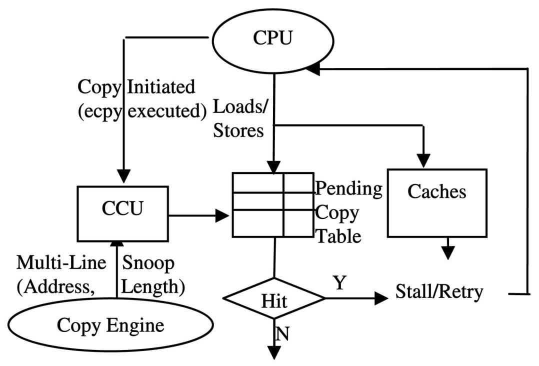


Fig. 9. Hardware support for asynchronous copies.

5.6.2 Copy Dependency

One more issue that needs to be addressed when using the copy engine is that of data dependency. For instance, once the copy command is initiated, the CPU continues to execute other instructions, which could be dependent on the copy source or the destination data. In order to detect these data dependencies, the CCU maintains the outstanding copy source and destination addresses in the pending copy table. Fig. 9 shows how dependency is detected and handled. Just as the subsequent loads and stores are compared to the pending memory transactions in the load/store queues of the CPU, they should also be compared against the copy commands in the pending copy table. This is especially critical when the copies are executed asynchronously. In this case, an external copy engine sends a multiline snoop to the CCU (as described earlier). Once the CCU receives the multiline snoop request, it enters the addresses and length into the pending copy table. This allows the CPU to perform dependency checks globally against copies initiated by all CPUs. Note that the size of the pending copy table is limited to ensure that the checking incurs low overhead. When a copy is issued and no entry is available in the table, the copy is either stalled or directly executed by the CPU.

6 Performance Evaluation

In this section, we present an in-depth evaluation of the impact of copy engines on TCP performance. We analyze copy engines along the following basic dimensions: 1) on-die versus off-die copy engines and 2) synchronous versus asynchronous copy execution. Our evaluation is based on a detailed simulation methodology, which is described in Section 6.1.

6.1 Simulation Methodology

Our copy engine simulation results are based on an execution-driven simulator SimpleScalar [27]. Since SimpleScalar had a simplistic memory subsystem model (based on a fixed latency calculation), our first task was to modify the cache/memory subsystem to a detailed event-driven model. Our cache/memory subsystem model implements 1) Miss Status Holding Registers (MSHRs) to limit the number of outstanding memory transactions, 2) a pipelined bus model to accurately model the latency and

TABLE 2
Architectural Parameters of Simulation

CPU	
CPU Clock rate	3.0 GHz
Peak issue, retire rate	4 instructions/cycle
Instruction window size	128
Functional units	2 integer arithmetic, 1 floating point
Cache	
L1 instruction/data cache	32 Kbytes, 4-way, 64-byte block
L1 cache hit time	2 cycles
L2 unified cache	1 Mbytes, 8-way, 64-byte block
L2 cache hit time	15 cycles
MSHR size	8
Bus	
system bus width	64 bits
system bus clock rate	800 MHz
Memory	
DRAM speed	400 MHz DDR 2
Average access time	80 ns (UMA/EMC), 60 ns (NUMA/IMC)
DRAM precharge time	15 ns
DRAM row access time	15 ns
DRAM column access time	15 ns

the queuing effect, and 3) a memory subsystem model that takes into account DRAM cycle times (row access, column access, and precharge) based on the page conflicts, the number of memory channels, and the Double Data Rate (DDR) memory technology [10]. The memory subsystem is configured to use an *open-page* policy. Our second major task was to add the hardware support required to enable synchronous and asynchronous copy engine models. This required the following: 1) addition of instruction support to the SimpleScalar ISA to emulate the *ecpy* instruction, 2) modeling of the communication between the CPU and the copy engine via the interconnect (if needed), and 3) modeling of the copy engine to generate coherent reads and writes to the memory subsystem. The final task was to calibrate the model with appropriate parameter values so as to simulate the delays observed in a realistic server platform. Although we do not have multiprocessor support in SimpleScalar, it should be noted that the delays modeled take into account the time taken for snooping all the processors in a four-way platform. A description of the architectural parameter values used for the server platform is provided below.

Our base system configuration is a four-way fetch/issue/commit MIPS microprocessor with an instruction window size of 128 entries and has two integer units, two load/store units, and a floating-point unit. We simulate a two-level cache hierarchy. The L1 I-cache and D-caches are 32 Kbytes in size, with 64-byte cache lines and four-way set associativity. The data cache is write-back, write-allocate, and nonblocking with two ports. The overhead to look up the copy pending table is the same as the L1 cache lookup. The L2 is a unified eight-way 1 Mbyte cache with 64-byte cache lines and a 15-cycle cache hit latency. The system bus is 64 bits wide (for data) and operates at 200 MHz (quad pumped), resulting in a bandwidth of 6.4 Gbps. The memory system is made up of two channels of DDR-II 400, contains 32 banks, and has a row buffer size of 2 Kbytes. The DRAM cycle times are assumed to be as follows: 3-cycle (15 ns) precharge penalty, 3-cycle (15 ns) row access, and 3-cycle (15 ns) column access. These configuration parameters are shown in Table 2. Variations on the parameters are explained where needed. For our platform configuration, we have an unloaded memory latency of approximately 80 ns (240 CPU cycles) for the UMA/EMC system architecture and a memory latency of 60 ns (180 cycles) for the NUMA/IMC system architecture. The latter latency assumes a uniform distribution of the memory accesses across two nodes in the platform and therefore includes a single-hop overhead to access the other nodes' memory subsystem. It should be noted that these memory latencies also include the delays taken to propagate the request and response via a realistic chipset. In addition to these latencies, the simulation takes into account the stalls experienced in the cache hierarchy (due to fully occupied MSHRs) and queuing delays experienced due to bus and memory traffic generated by the application.

The TCP/IP processing workload that we use is derived from the FreeBSD stack [20] and performs receive-side processing. To simulate different types of network traffic, we drive the TCP/IP stack with three different packet traces. Each trace contains 100,000 packets, with a fixed packet payload size of 512, 1,024, and 1,400 bytes, respectively. These packet sizes were chosen to represent the typical receive traffic in a Web server, e-mail server, and database server configurations.

6.2 Summary of Copy Engine Simulation Configurations

We present and analyze simulation results for three copy engine configurations: 1) SYNC_NLB—synchronous copy engine with no load bypass, 2) SYNC_WLB—synchronous copy engines with load bypass, and 3) ASYNC—asynchronous copy engines, which assume load bypass. As mentioned earlier, load bypass basically allows cache misses generated by the CPU to be interleaved within the copy engine requests and therefore does not have to stall until all outstanding copies are completed. We compare these three copy engine configurations with the basic system configuration without a copy engine (BASE). All three traces are run through the TCP stack in these four configurations (BASE, SYNC_WLB, SYNC_NLB, and ASYNC) for off-die copy engines. Subsequently, for space considerations, we focus primarily on the trace with 1 Kbyte packets when comparing the on-die copy engine versus the off-die copy engine and when studying the sensitivity of the copy engine to various architectural parameters and optimization. All of the data is shown in the form of percentage of execution time reduction as compared to the base case.

6.3 Performance Benefits of Off-Die Copy Engines

Fig. 10 shows the execution time reduction with the copy engine as compared to a BASE system running with a MIPS-like processor running in In-Order (IO) and Out-of-Order (OOO) execution modes. Although our primary focus is the system with an OOO MIPS-like core, we compare against an IO core to show how poorly the copy may perform if the execution core is unable to extract parallelism by looking ahead in the instruction window. As expected, the benefits of a copy engine are significantly greater in an IO system than in an OOO system. For instance, when processing 512-byte packets in an IO system, the use of a copy engine reduces the execution time by 43 percent to 58 percent based on the copy execution model used. When processing the same-sized packet in an OOO system, the use

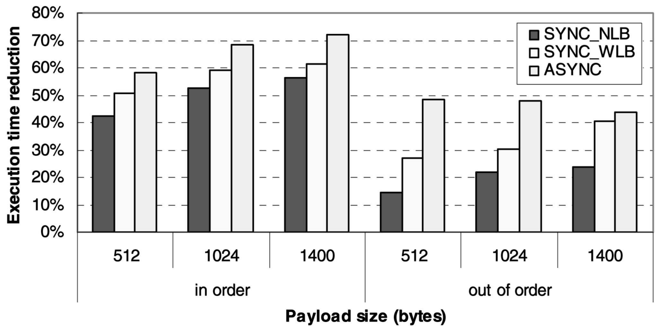


Fig. 10. Performance benefits of off-die copy engine in the UMA/EMC architecture.

of a copy engine reduces the execution time by 15 percent to 48.5 percent. Based on these results, we also notice that the difference between the synchronous engine and the asynchronous engine performance is less significant in an IO system than in an OOO system. The reasoning for this is that the major benefit of a copy engine in an IO system is the reduction in the copy latency itself, whereas a significant portion of the benefit in an OOO system is due to the computation overlap.

To understand the underlying reasons for the benefits of copy engine, we break down the execution time reduction caused by the copy engine into two parts: 1) copy speedup—this is due to the copy engine moving data at a faster rate and 2) the computation overlap—some computation can be executed in parallel with the copy operation. As confirmed in Table 3, in an IO execution system, most of the improvement is from the faster copies. This is because the copy performance, when executed by an IO CPU, is really poor. In addition, the amount of computation overlap is limited because of the IO restriction placed on execution. Therefore, even with ASYNC copy engines, the number of the instructions that can be executed in parallel with the copy is very limited.

On the other hand, for an OOO system, the three copy execution models have different contributions to the overall execution time reduction. With SYNC_NLB, the improvement is mostly due to faster copies when employing copy engines. Since this approach does not allow subsequent load instructions to bypass the copy command, only a limited number of instructions can be executed in parallel with the copy operation. Those instructions that have data dependency on the load instructions have to wait until the copy completes. SYNC_WLB improves upon this by allowing CPU-issued load instructions to bypass the copy command so that more instructions can be executed in parallel with the copy. For 512-byte packets, the computation overlap is increased from 2.5 percent (out of 14.6 percent) for SYNC_NLB to 19.6 percent (out of the 27.1 percent) for SYNC_WLB. However, SYNC_WLB is still limited by the fact that the number of subsequent instructions that can be executed cannot exceed the size of the instruction window and the load/store queue (128 and 64 in our case).

ASYNC alleviates this limitation by allowing the copy instruction to retire sooner than the actual completion of the copy command. This increases the computation overlap from 37 percent (for SYNC_WLB) to 41.1 percent (for ASYNC). As we increase the packet size from 512 to 1,400 bytes, we can see the improvement in copy speeds and reduction in overlap for all three schemes. The former observation is obvious since the percentage of the copy latency in the total execution time is higher with larger payload sizes. Thus, faster copies contribute more to the improvement. The latter observation is for the same reason: The percentage of the computation is smaller with larger payload sizes. In summary, we believe that an asynchronous execution model is critical to providing sufficient performance benefits, even with larger payload sizes.

6.4 Benefits of On-Die Copy Engines

In Section 6.3, we evaluated the off-die copy engine for the UMA/EMC architecture. In this section, we present the benefits of an on-die copy engine for the NUMA/IMC architecture and compare it to the off-die copy engine benefits. The simulation results (for 1 Kbyte packet processing) are shown in Fig. 11. The percentage reduction in execution time is shown as compared to the BASE UMA/EMC system. It can be observed that the NUMA/IMC system performance is better than the UMA/EMC system by about 10 percent because of the reduction in memory access latency (due to integrated memory controller). When enabling copy engines, we find that the benefits of synchronous copy engines are about the same for both architectures. However, the performance of the asynchronous on-die copy engine is lower than that of the off-die copy engine. This occurs because the computation is completely overlapped with the copy operation, which leads to the copy engine becoming the bottleneck. Since the copy engine access time is increased in the on-die system, the benefit is reduced.

TABLE 3
Factors Affecting Execution Time Reduction

Payload Size (bytes)	CE Modes of operation	In order			Out of order		
		Overall	Faster Copies	Overlap	Overall	Faster Copies	Overlap
512	SYNC_NLB	42.5%	40.7%	1.8%	14.6%	12.1%	2.5%
	SYNC_WLB	50.5%	37.4%	13.1%	27.1%	7.5%	19.6%
	ASYNC	58.1%	37.4%	20.7%	48.5%	7.5%	41.1%
1024	SYNC_NLB	52.5%	51.4%	1.1%	21.9%	20.4%	1.5%
	SYNC_WLB	58.9%	50.5%	8.4%	30.1%	17.4%	12.7%
	ASYNC	68.2%	50.5%	17.7%	48.0%	17.4%	30.6%
1400	SYNC_NLB	56.3%	55.4%	0.9%	23.8%	22.7%	1.1%
	SYNC_WLB	61.2%	54.5%	6.7%	40.3%	32.0%	8.3%
	ASYNC	72.1%	54.5%	17.6%	43.9%	32.0%	11.9%

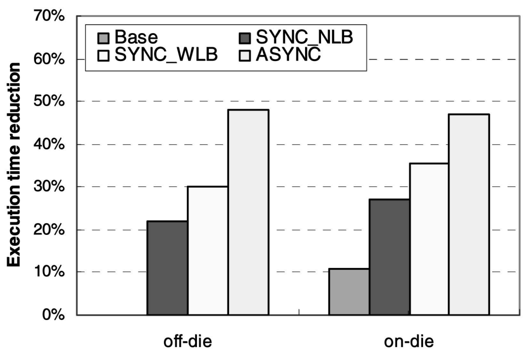


Fig. 11. Performance benefits of the on-die copy engine.

When considering on-die copy engines in the NUMA/IMC architecture, we also evaluate the benefits of early notification for the asynchronous execution model. Here, the early notification is sent after receiving an acknowledgment from the processors in the platform that it has observed the outstanding copy. The processors can observe the copy in two different ways: 1) by performing a snoop on the copy regions and responding (after snoop) and 2) by entering the addresses in the pending copy table and responding. Table 4 shows a comparison of these two modes of operation. The first row in the table shows the time taken for the acknowledgment to be received by the copy engine. It should be noted that this is considered only for asynchronous copy engines since the snoop time can be overlapped with the memory access time in the case of synchronous copy engines. It should also be noted that this is not considered for UMA/EMC because the bus is a global medium and the copy is globally observed when the request is placed on the bus by the requesting processor. For asynchronous copy engines in NUMA/IMC, the BASE case requires the snoop to return within 30 ns so as to overlap with the communication between the processor and the copy engine. However, if the multiline snoop needs to be completed at the processor caches, then the time taken can be much longer (as shown in the second column). The *block_num* is the number of cache lines that are invalidated. For a 1 Kbyte packet, the snoop time is 84 ns. The effect of this additional snoop time before the *ecpy* instruction can be retired translates to a reduction in performance benefit of about 2 percent (from 40.7 percent to 37.8 percent). As a result, either approach can be employed from a performance perspective.

6.5 Impact of Memory Subsystem

We also look at the impact of the memory system on the performance. In Fig. 12, we increase the memory bandwidth from 3.2 GHz to 6.4 GHz and show that the execution time reduces significantly in all four modes of operation for off-die copy engines. This is expected because, even with the help of the copy engine, the bottleneck still remains in the memory system. Therefore, faster memory gives better performance. In Fig. 13, we decrease the memory access latency from 110 to 80 ns. We can see that the memory access time has less impact on the copy engine than the BASE case. For instance, the execution time of the BASE is reduced by about 10 percent and 18 percent when the latency is reduced to 95 and 80 ns, respectively. However, with SYNC_NLB, the execution time reduction changes by hardly 2 percent (from 33.7 percent to 36.1 percent). The difference is even less for SYNC_WLB and ASYNC copy engines (about 1 percent and 0.2 percent, respectively).

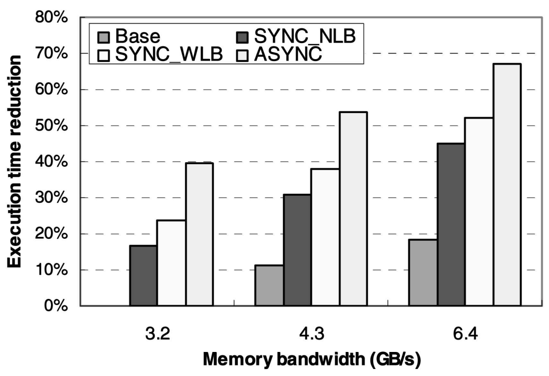


Fig. 12. Impact of memory bandwidth.

6.6 Effects of CPU Frequency

We vary the processor frequency and measure its impact on the off-die copy engine performance. The results are normalized to the base case when the CPU frequency is 2.0 GHz. As shown in Fig. 14, the execution time reduction is 7.2 percent and 10.1 percent for the BASE case when the frequency is increased to 3 and 4 GHz, respectively. This benefit is entirely due to the computation speedup by the faster CPUs as the memory subsystem is not scaled at all. However, the benefit of faster CPUs declines as we start using synchronous and asynchronous copy engines. There continues to be a benefit due to faster copies in copy engines, but the amount of overlap possible is reduced in absolute time (in nanoseconds, not cycles). One can observe in the figure that faster CPUs have almost no effect on the ASYNC mode of execution because the computation is already (100 percent) overlapped with the copy operation.

TABLE 4
Early Notification on ASYNC

	Early Notification Before Snoop	Early Notification After Snoop-
Ack time	<= 30ns	20ns + 2ns *2*block_num
Improvement	40.7%	37.8%

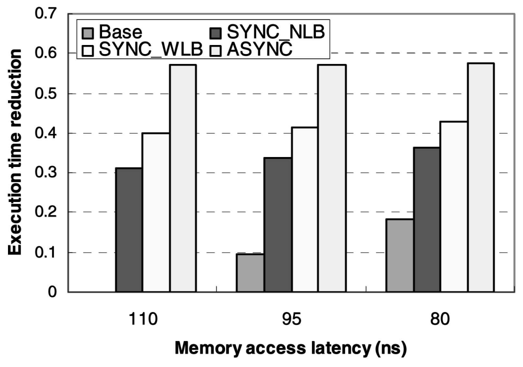


Fig. 13. Impact of memory latency.

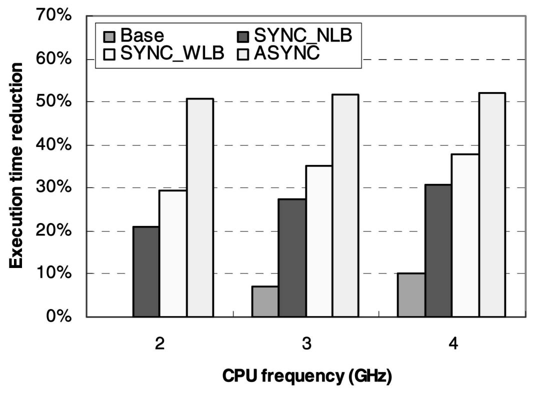


Fig. 14. Impact of the CPU frequency.

Here, the copy engine becomes the bottleneck and thereby determines the execution time.

6.7 Benefits of Cache Usage during Copies

Until now, most of the above simulation results were based on the copy engine issuing reads and writes directly to memory. Given that we have an on-die copy engine, it is possible for the destination to be written directly into the cache by the copy engine. This has the potential to speed up the copy operation since it can be performed faster than the memory write and also has the potential to reduce subsequent cache misses if the application touches the destination soon enough. Although we do not evaluate the effect of the application touching the data, we have simulated the effect of writing the destination to the cache. The simulation results are shown in Fig. 15. The results show that the synchronous copy engine provides an additional 20 percent reduction in the execution time if the destination is written to cache. The additional reduction for an asynchronous copy engine is about 13 percent. This is because less of the copy operation time is overlapped in the case of synchronous operation as compared to the asynchronous operation.

7 Conclusions and Future Work

In this paper, we described the bulk data movement problem in servers and studied its impact on TCP/IP processing, which is a workload common to many network-intensive server applications. Having reviewed existing techniques for latency hiding during data movement, we showed that most of these are either ineffective or have limited applicability. To address this problem directly, we proposed the use of copy engines that can be tightly integrated into the server platforms.

We then discussed the architectural considerations for integrating copy engines and identified two major dimensions: 1) on-die placement versus off-die placement and 2) synchronous versus asynchronous execution models. We then described the implementation options for these copy engine solutions and the associated hardware support required in the platform, including CPU support, interconnect support, copy engine design, and coherence/synchronization requirements.

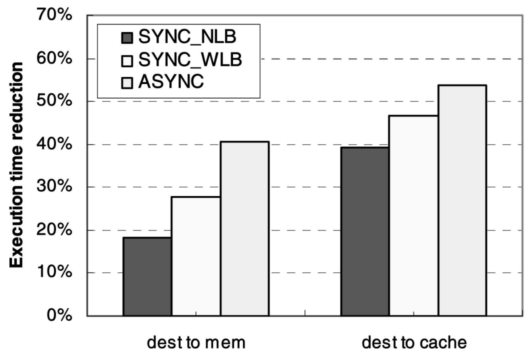


Fig. 15. Impact of using caches in copies.

Finally, we modeled the copy engine solutions by extending an execution-driven simulator and showed that the performance benefits of the proposed copy engines are significant. Our analysis of the simulation results also showed that asynchronous on-die copy engines are desirable from a performance standpoint. In addition, we also showed that the proposed copy engines may even be able to take advantage of the local processor's cache to provide additional speedup.

We believe that the integration of copy engines in server platforms has significant potential. In future work, we plan to evaluate the benefits of copy engines for a wider range of applications. We also plan to extend our analysis to other bulk data operations such as memory initialization, parsing, encryption, and others. We believe that the basic framework that we described in this paper can be easily extended to accommodate these other frequently occurring operations.

References

[1] AU-S3000, http://vodka.auroravlsi.com/website/product_briefs/au-s3000_brief.pdf, 2003.
[2] L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th ACM Int'l Symp. Computer Architecture,* June 2000.
[3] J. Brustoloni, "Interoperation of Copy Avoidance in Network and File I/O," *Proc. IEEE INFOCOM '99,* pp. 534-542, Mar. 1999.
[4] J. Brustoloni and P. Steenkiste, "Effects of Buffering Semantics on I/O Performance," *Proc. 1996 Symp. Operating Systems Design and Implementation (OSDI II),* pp. 277-291, Oct. 1996.
[5] D. Callahan et al., "Software Prefetching," *Proc. Fourth ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* Apr. 1991, *Multithreading,* vol. 35, Mar. 2003.
[6] J. Cater et al., "Impulse: Building a Smarter Memory Controller," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture (HPCA 5),* Jan. 1999.
[7] T. Chen, "An Effective Programmable Prefetch Engine for On-Chip Caches," *Proc. Micro-28,* pp. 237-242, Dec. 1995.
[8] D. Clark et al., "An Analysis of TCP Processing Overhead," *IEEE Comm.,* pp. 23-29, June 1989.
[9] D. Clark and D.L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '90),* pp. 200-208, Sept. 1990.
[10] DDR Memory, http://www.intel.com/technology/memory/, 2004.
[11] A. Earls, "TCP Offload Engines Finally Arrive," *Storage Magazine,* Mar. 2002, http://storagemagazine.techtarget.com/.
[12] A. Foong et al., "TCP Performance Analysis Revisited," *Proc. IEEE Int'l Symp. Performance Analysis of Software and Systems (ISPASS '03),* Mar. 2003.

[13] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* second ed. Morgan Kaufmann, 1995.

[14] Y. Hoskote et al., "A 10GHz TCP Offload Accelerator for 10Gb/s Ethernet in 90nm Dual-Vt CMOS," *Proc. IEEE Int'l Solid-State Circuits Conf. (ISSCC '03),* 2003.

[15] "IA-32 Intel Architecture Optimization Reference Manual," http://www.intel.com/design/Pentium4/documentation.htm, 1999.

[16] iSCSI, IP Storage Working Group, Internet Draft, work in progress.

[17] S. Makineni and R. Iyer, "Performance Characterization of TCP/IP Processing in Commercial Server Workloads," *Proc. Sixth IEEE Workshop Workload Characterization (WWC-6),* Oct. 2003.

[18] S. Makineni and R. Iyer, "Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor," *Proc. 10th Int'l Symp. High-Performance Computer Architecture,* Feb. 2004.

[19] D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology J.,* Feb. 2002.

[20] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman, *The Design and Implementation of the 4.4BSD Unix Operating System.* Addison-Wesley Publishing, 1996.

[21] J. Mogul, "TCP Offload Is a Dumb Idea Whose Time Has Come," *Proc. Symp. Hot Operating Systems (HOT OS),* 2003.

[22] D. Patterson et al., "A Case for Intelligent DRAM: IRAM," *IEEE Micro,* Apr. 1997.

[23] J.B. Postel, "Transmission Control Protocol," RFC 793, Information Sciences Inst., Sept. 1981.

[24] "Pentium(R) Pro Family Developer's Manual: Specifications," vol. 1, http://developer.intel.com/design/archives/processors/pro/docs/242690.htm, 1996.

[25] RDMA Consortium, http://www.rdmaconsortium.org., 2003.

[26] Remote Direct Data Placement Working Group, http://www.ietf.org/html.charters/rddpcharter.html, 2005.

[27] SimpleScalar LLC, http://www.simplescalar.com, 2000.

[28] The SPECweb99 Benchmark, http://www.spec.org/osg/web99/, 1999.

[29] "Accelerating Core Networking Functions Using the UltraSPARC VIS[tm] Instruction Set," Sun Microsystems Laboratories http://www.sun.com, 1996.

[30] M.N. Thadani and Y.A. Khalidi, "An Efficient Zero-Copy I/O Framework for UNIX," Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, May 1995.

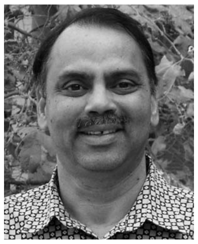
[31] *The TPC-C Benchmark,* http://www.tpc.org/tpcc/, 2005.

[32] *The TPC-W Benchmark,* http://www.tpc.org/tpcw/, 2005.

[33] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 392-403, June 1995.

[34] T. Ungerer, B. Robic, and J. Silc, "Multithreaded Processors," *Computer J.,* vol. 45, no. 3, 2002.

Li Zhao received the PhD degree in computer science from the University of California, Riverside, in 2005. She is a senior engineer in the System Technology Laboratory at Intel. Her research interests include computer architecture, network computing, and performance evaluation. She is a member of the IEEE.



Laxmi N. Bhuyan has been a professor of computer science and engineering at the University of California, Riverside, since January 2001. Prior to that, he was a professor of computer science at Texas A&M University (1989-2000) and program director of the Computer System Architecture Program at the US National Science Foundation (1998-2000). He has also worked as a consultant to Intel and HP Labs. Dr. Bhuyan's current research interests are in the areas of computer architecture, network processors, Internet routers, and parallel and distributed processing. He has published more than 150 papers in related areas in reputable journals and conference proceedings. He has served on the editorial boards of *Computer*, the *IEEE Transactions on Computers*, the *Journal of Parallel and Distributed Computing*, and the *Parallel Computing Journal*. He is the editor-in-chief of the *IEEE Transactions on Parallel and Distributed Systems*. Dr. Bhuyan is a fellow of the IEEE, the ACM, and the AAAS. He is also an ISI Highly Cited Researcher in 'Computer Science.

Ravi Iyer received the PhD degree in computer science from Texas A&M University. He is currently a principal engineer in the Systems Technology Laboratory at Intel. He is an associate editor for the *IEEE Transactions on Parallel and Distributed Systems* and has been actively involved in several conferences and workshops. His current research is focused on large-scale chip multiprocessor (CMP) architectures and technologies. He has published more than 60 papers in the areas of computer architecture, server design, network protocols/acceleration, workload characterization, and performance evaluation. He is a member of the IEEE.

Srihari Makineni received the master's degree in electrical and computer engineering. He is a senior software engineer in the Corporate Technology Group at Intel. He joined Intel in 1995 and has worked on videoconferencing, multimedia streaming, Web/e-commerce applications, and system and server management technologies. His areas of interest include networking and communication protocols and computer architecture. His current work is focused on developing techniques for accelerating packet-processing applications on IA-32 and Itanium Processor Family (IPF) architectures.

Donald Newell joined Intel in 1994 and currently works in the Systems Technology Laboratory. He has spent most of his career working on networking and systems software for server platforms and real-time systems. He has worked on a number of emerging technologies at Intel. This includes leading the group that developed Intel's frameworks for media streaming over the Internet and to support data broadcast for digital television (DTV). He chaired the Advanced Television System Committee (ATSC) work on data broadcast in DTV and was a coauthor of *IETF RFC 2429*. He and his group were also key contributors to one of the recently announced *T's-Intel I/O Acceleration Technology (Intel I/OAT). Currently, he leads a group working on CTG's many-core program.